Research report 129

FORMAL DESCRIPTION OF REAL-TIME SYSTEMS: A REVIEW

Mathai Joseph, Asis Goswami

(RR129)

Work on the formal description and analysis of real-time systems has followed two paths. On one side, it has considered the specification of real-time systems, the design of language constructs for real-time programming, and semantic models to describe the properties of real-time programs. On the other side, there has been a large body of work analysing the performance of real-time systems in terms of the feasibility of schedules and the use of resources, especially in relation to hard-real-time problems. This paper reviews some of this past work and examines how real-time computations are modelled in a proof-theoretic framework and in scheduling analyses. The object of this review, and indeed of other contemporary work in the field, is to examine whether it is possible to relate issues of correctness and performance, eg interpreting the semantics of execution of real-time programs in terms of scheduling solutions to real-time problems.

This work was supported by research grant GR/D73881 from the Science and Engineering Research Council.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

August 1988

1 Introduction

A program P is said to be correct when it *satisfies* a given specification S. This satisfaction relation must be established using an abstract model of the language used in the program, and this is typically represented by the axioms and rules of a proof system.

The set of program properties described in the specifications determines the kind of proof system needed to verify the correctness of programs. Depending on the kind of specification used, there may be need for proof systems for *partial correctness*, *total correctness*, total correctness of concurrent programs without *deadlock*, and so on. Adding to this set of properties will often require changes in the axioms or rules defining the language constructs; this will be the case if the new properties are not provable in the existing proof system. For example, *fair termination* [11] of guarded commands [7] is not provable unless the proof system includes assumptions about fairness in the selection of guards.

If the proof system for a language takes account of the availability of computational resources such as, say, the number of processors and their speeds or the extent of main memory, a new class of program properties that give measures of resource usage can be defined. Time as viewed by the environment in which a program executes is a significant factor in many applications and the program properties relating to time are called its *real-time* properties. Unlike the usual *safety* and *liveness* properties which are defined over the entire execution of a program, real-time properties are defined over particular time intervals. Verifying these properties involves determining the time of completion of actions in the program, and this in turn may depend on the availability of computing resources. If the resources can be assumed to place no restriction on the computations, e.g. if processing takes no time, a relatively simple synchronous language model (like that of SIGNAL [12]) can be used in which each computation takes place as soon as it is needed, and the problem of resource allocation becomes trivial. If, on the other hand, resources are limited, the time needed for a computation can depend on the availability of resources and a more complex language model is needed to represent the allocation of resources to computations.

Resource scheduling is also related to what is called the *performance* of a program, which is a description of its real-time behaviour at some level of abstraction. Program performance can be studied analytically, or by measuring the actual performance during an execution (in both cases, however, a more abstract form of the program than its full text is usually used). Performance studies are concerned with issues such as whether a schedule of resources is feasible, whether an implementation will permit a feasible schedule of resources, etc.

The correctness and performance of a real-time program are, in principle, unifiable. Performance studies take into account implementation details such as the number of processors in the machine and the structure and bandwidth of its communication network. Program verification systems, on the other hand, usually abstract away from implementation considerations and concentrate on program structure. Use of a higher level of abstraction in program structure for performance

studies is only possible because of assumptions made about the computation times of program components at lower levels of decomposition. Some early work on the proof-theoretic analysis of computation times of programs [44,46] suggests that proof systems for untimed program properties can be extended to reason about the computation times of programs. What is needed for the unification of correctness and performance of real-time programs is a way of describing, within a proof-theoretic framework, the availability of resources and performance measures such as the existence of a feasible schedule. In this paper we review some past work on formal models for real-time programs and on scheduling for real-time systems; the review is not intended to be exhaustive and references have been chosen to be illustrative. We shall be concerned primarily with concurrent and distributed real-time programs as these are most common in practice. We will examine how resources are modelled within a proof-theoretic framework and in scheduling algorithms using a system model. Thus, on one side the review covers work on the specification and semantics of real-time programs, while on the other it summarizes the results from performance models and describes various real-time scheduling techniques. In conclusion, we stress the need for a single framework for real-time programs in which it is possible to see the relation between issues of correctness and performance, eg between the semantics of execution of concurrent programs and solutions to real-time problems.

2 Program Performance and Correctness

The performance of a program can be studied by measuring the resources that it uses in a specific execution model; thus, performance studies may estimate, for a particular system, the processor time and memory space used, or consider how the execution may be affected by changes in the computer system on which it executes. Program performance can be estimated in several ways: by simulation, by measurement using software or hardware probes during an actual execution, or by using a stochastic model of the program. It can also be studied analytically in terms of the structure of the program, in much the same way that the properties of a program are determined.

The earliest work relating program performance with proof techniques was by Wegbreit [46] who extended Floyd's method of inductive assertions [8] by assigning probabilities for the state of the program at points during its execution. Given probabilities for the different possible input states of the program, the method allowed the probabilities for the final states, and thereby the mean computation time, to be computed. Proof of performance was then similar to proving the correctness of the program by inductive assertions; however, the method applied to *average* computation times, not the worst-case times needed for real-time program analysis.

The technique of adding probabilities about performance to an annotated proof outline of the program can of course be extended to apply to other proof techniques. But, following a different approach, Mary Shaw [44] added exact cost computations to predicate transformers [7] for a simple, deterministic, Pascal-like language to allow reasoning about cost preconditions to be done in conjunction with determining weakest preconditions. The cost computations could be made with considerable

detail, enough to take account of the overheads of statement execution. But it is not easy to sustain such exactness throughout even a simple language because for cases such as **if** commands (where there is no information on which alternative will be selected) and **while**-loops (where in general the exact number of iterations cannot be specified for arbitrary input data) the cost would be need to be computed only as an upper bound. This could lead to an unrealistic estimate of the cost, especially for nested commands, or for a language with non-deterministic constructs. Note that even for simple sequential composition, the actual upper bound to the cost of computation of two sequential **if** commands may be smaller than that obtained by by adding together the separate upper bounds for each **if** command when, for example, the worst-case alternatives of the two **if** commands are mutually exclusive.

Haase [13] discussed the real-time performance of programs executed on a multiprocessor. He used an extended language with parallel alternative and repetitive constructs and showed how, using an explicit time variable, predicate transformers can be defined to specify worst-case timing. Since parallel execution can lead to arbitrary interleavings of commands, the method must consider the transitive closure of *all* interleavings of parallel sub-commands. The analysis can thus become very complex and he recommended the use of simple programs and system architectures; however, since there is a global time variable, the set of *possible* interleavings could be determined by considering their time-ordering, and this could help to simplify the analysis.

From the point of view of determining real-time properties, there are some disadvantages in associating program performance too closely with the proof of other properties. Every command in a program has some irreducible requirements of resources and the performance of the program will be determined by the way in which the commands are scheduled, not just by the resource requirements of each command. There may be several different ways of scheduling the same program to meet its real-time specification, and there may be several different complements of available resources for which this may be done. In this respect, scheduling is akin to other implementation decisions and should therefore be considered at that level. To make the distinctions between different kinds of program properties clear, we shall base the rest of this review on a framework in which requirements, specifications, program semantics and scheduling properties can be viewed as a hierarchy.

A Framework for Description

Let us define a real-time system as a concurrent system whose processes work in conjunction with a set of external processes in the real world. The environment of the real-time system consists of these external processes and the events they send to the system; the internal processes of the system must receive these events and respond by sending other events to the environment. A real-time system may be called *embedded* if is physically enclosed in its environment, for example like a system on-board an aircraft; an embedded system may have special reliability requirements but otherwise its problems are like those of any real-time system.

The events produced by the external processes induce timing constraints over

the responses to be produced by the system. The external events may be repeated at regular intervals or they may appear sporadically, ie they may be *periodic* or *aperiodic*. For example, a pyrometer process may send an event to a system aperiodically, only when it records a high temperature, and the system may be required to respond by closing a fuel inlet valve within a limited time interval. Another system may be required to periodically accept readings from a pyrometer and to close a fuel valve within a limited time after it receives a high temperature reading. A periodic incoming event must be serviced by the periodic execution of some internal process and will impose a regular base load on the system. Aperiodic events will not appear regularly but the system must nevertheless be able to respond to them at any time; this often makes it easier to treat aperiodic events as periodic events with some specified minimum time between repeated occurrences.

A real-time system can be described at several different levels: for example, in terms of its external requirements, its formal specification, its implementation in some programming language and its execution on some particular system. But the distinction between a specification and an implementation is often relative in that what is called the implementation at one level may be the specification for the next level: when this is true, the system has the attractive property of a hierarchical design.

The first step in specifying a real-time system is to define the *requirements* of the environment. These *requirements* can be defined as a set of relations between the external events and the desired responses and their distribution in time of events and responses. The relation between external events and responses may be simple, with one response required for each external event, or quite complex, for example requiring a set of responses for different combinations of events distributed in time. It is easy to see that the task of defining the requirements for a large real-time system in this way can be difficult because the full set of relations between external events and responses must be enumerated.

Once the requirement definition is available, the formal *specification* of the real-time system can be developed. The goal of this specification is to define a class of programs that will satisfy the requirements. The specification must therefore define the actions to be performed by the program and the timing constraints that apply. A real-time specification differs from other formal program specifications (such as Z [14], CSP [18], VDM [24], or CCS [35]) in associating timing with actions. Such a specification must be supported by a semantic model which incorporates time.

The *implementation* of a system must satisfy its specification and, in the case of a real-time system, this means that the program must satisfy the functional requirements and that its implementation on a particular system must meet the timing requirements. There are thus two parts to the implementation and they can be considered separately. A program will meet its specification if it is derived from the specification by formal transformation, or if the properties of the program (determined by the proof rules for the language) can be shown to satisfy the specification. Defining the implementation of the program requires describing its translation for machine execution and defining the resources needed on the system on which it will run, as both of these will affect the timing characteristics of its execution.

3 Requirements, Specifications and Models

Formal specification of the timing characteristics of real-time systems was first proposed by Bernstein and Harter [1]. In this approach, the specification of a program is an assertion in a linear-time temporal logic about the execution of the program. In the underlying computation model (or program semantics), an execution of a program is described by the infinite sequence of states entered by the program during the execution; in a terminating execution, the final state is repeated indefinitely. Each (indivisible) operation of the program causes a state transition.

The model uses a global time domain which is the set of real numbers. Every state in the semantic description of a program has associated with it the time at which it is entered. Real time is introduced into the specification language by extending the "temporal implication" to have time bounds. The formula

$$A_1 \overset{<n}{\rightsquigarrow} A_2$$

asserts that if execution reaches a state satisfying A_1 then it will reach a state satisfying A_2 in less than n units of time. On the other hand, the formula

$$A_1 \overset{\geq n}{\rightsquigarrow} A_2$$

asserts that if execution reaches a state satisfying A_1 then A_2 will not be true in that state and the execution will not reach a state satisfying A_2 within the next n units of time.

The proof system is limited to establishing specific real-time safety properties. It uses an interleaving semantics of programs and, therefore, can be applied most easily to uniprocessor implementations. The assumptions made about the execution times of program commands and restrictions on the kind of permitted interleaving make it difficult to extend this proof system to a more general model of real-time programs.

Another model, also based on linear-time temporal logic, was proposed by Koymans et al [29]. The underlying semantics is state-based and the states are timed with a global clock. However, unlike the previous model [1], the time domain here is discrete and no assumptions are made about the execution times of commands. The usual temporal logic concept of a 'location' predicate [30] is generalized to model dynamic process creation, and the temporal operator $\mathcal{U}_{=t}$ (*strong until* in time t) is introduced to express and verify liveness properties of real-time programs.

Although this proof system provides a timed-model for asynchronous communication and time-out, it does not give permit timing of other programming constructs such as assignment, sequential and parallel composition, etc. Thus, the applicability of this proof system is restricted to certain segments of a program. Also, the semantics is based on the assumption that parallel processes are never interleaved – a variant of the so-called *maximal parallelism* assumption [42]. There are many implementations where this will not hold, particularly when processes are created dynamically.

The temporal logic proof system of Koymans [27] uses a dense order for the time domain and extends both the 'until' and the 'since' operator to have times associated with them. The proof system is more powerful than its predecessor because more safety and liveness properties can be proved for general message passing systems. However, it has some of the same limitations.

The timed model of distributed systems proposed by Shankar and Lam [43] uses both the notions of states and events. Each process has a set of state variables and a set of events. An event is modelled as a state-transformer together with its enabling conditions. Safety and liveness properties are verified in a first order logic extended by the temporal operator "leads-to" [31]. However this operator does not have explicit time associated with it. Instead, each process is assumed to have a set of 'local timers' which are distinguished state variables. These timer variables take values from the domain of the natural numbers and can be enabled (reset to 0) or disabled by events. The 'local time event' of each process increments the enabled timers of the process to their next higher values. A local timer of one process cannot affect the behaviour of a local timer of another process. Thus it would not be possible to describe code execution time in this model unless the local time events of the processes are coupled to termination events associated with individual program commands and this would, in any case, make the proofs rather complicated.

The Real-Time Logic (RTL) of Mok [23] is based on a model of distributed systems that uses a predefined set of event names and a set of primitive actions. All primitive actions terminate and their execution times are assumed to be known. Each action has two associated events – the 'initiation' event and the 'termination' event. RTL uses a notational device, called a *state predicate*, to describe the truth value of a boolean system attribute during an 'interval' (represented by pairs of event occurrences). Each state predicate describes two events: the two transition events that change the value of the corresponding system attribute to **true** and **false** respectively.

Events are central to the calculus of RTL because actions and state predicates are also transformed into events. Reasoning about a real-time system is based on assertions about the occurrence of events in the system. An 'occurrence function' maps an occurrence of an event into the time of that occurrence. The time domain is the set of nonnegative integers.

RTL is used for reasoning about the safety properties in the requirement specification of real-time systems given in an event-action model [23,40]. A specification consists of a set of primitive and composite actions of the system, a set of known events, a set of state predicates and the timing constraints on the actions. The actions interact in a simple way so that a partial order of the primitive actions of a system is easily derived from the text of its description. In fact, a specification in the event-action model can be mechanically transformed into a set of RTL formulae. The timing constraints translate into restrictions over the 'occurrence function' of RTL. A given safety property holds if there is no mapping of event occurrences into time values which is consistent with the negation of the safety property and the specified restrictions on the 'occurrence function'.

The modelling of a real-time system as a partial order of computational actions may be adequate for the purpose of requirement description in many applications, but it is not useful for developing an implementation in a realistic programming language with command combinators such as iteration, recursion, or parallel composition with variable sharing or message communication. There have been some recent proposals to incorporate real-time in Hoare's CSP [18,17] and variants of CSP equipped with constructs to express the real-time behaviour of programs [10,20,21,28,41,49]. Usually, the construct "**wait** d" or "**delay** d" is used to suspend execution of a program for d units of time. In some cases [20,21,28], this construct is used as the alternative with least priority in a composite guarded command; time-out mechanisms can be expressed in this way.

The first real-time model for developing specifying and verifying programs in CSP-like languages was proposed by Koymans et al [28]. This is a denotational model which modifies and extends the linear-history semantics of CSP [9]. The execution model is based on a maximal parallelism model of processes. The denotation of a process is a nonempty prefix-closed set of (state,history) pairs. Each history is a finite sequence of *bags* of communication assumption records leading to the state associated with the history. A communication assumption record describes whether communication between two particular processes is possible and, if so, specifies the value that is communicated. Bags of such records are used instead of sets in order to model simultaneity in communication. The internal actions in a process are modelled by empty bags. Time in this model is global and discrete. Each atomic action takes a unit time for its execution. The elements of an history are implicitly timed in terms of their positions in the history. Since internal actions are represented in the histories, time can conveniently be modelled in this way.

It is of great value for a specification system to be *syntax-directed* or *compositional* [50,51] so that the specification of a composite command can be constructed from the specification of the components of that command without making use of the internal details of these components. In the Koymans model [28], compositionality is achieved by making the semantics sufficiently detailed although some of the information in the semantic description of the system being modelled will not be of interest to an external observer of the system. The information content of the specification can therefore be partitioned into two parts – one part about the *observable* entities of the system and the other about the non-observables. The choice of the observable entities depends on the applications of the semantics. Given a particular set of observables, it desirable to obtain a *minimal* set of non-observable entities to make the semantics compositional; if this can be done, then we achieve *full abstraction* of the semantics (with respect to the set of observable entities) [15]. Viewed in this light, the semantics of the model of [28] is not fully abstract.

A later version [21] describes a fully abstract semantics for an Occam-like language [22]. The observable entities of a process in this model are its initial state, its final state (if it terminates), the values communicated to and from the process, and the times of these communications. These entities are adequate for describing the real-time behaviour of communicating processes. The periods during which the processes wait for communication are the non-observables needed to make the semantics fully abstract.

These two semantic models are similar in the way they capture the real-time behaviour of proceses communicating by synchronous message passing. Another semantic model, which has borrowed many of its basic ideas from these two models has been used to develop a compositional proof system for an Occam-like language [20]. Time in this semantic model is still global, but it ranges over a dense total order having a least upper bound and a greatest lower bound. The execution time of an atomic command lies in an interval which depends on the program state.

Hooman's specification language [20] uses a global variable *time* to refer to the (conceptual) global clock. The proof system is in the *assumption-commitment* style [26,36,51]: the correctness formulae are of the form

$$(A,\ C)\ :\ \{p\}\ L\ \{q\}$$

where L is a command, p and q are Hoare-like pre- and post-conditions [16], A is an assumption describing the (expected) behaviour of the environment of L and C is a commitment of C provided that the environment satisfies the assumption. The environment of a process consists of its communication partners and is syntactically determinable. The rule of parallel composition of commands is based on ensuring that the assumptions of each process about its communications correspond to the commitments of the other processes towards these communications.

Informally, if L starts execution in a state satisfying p then the following is true at all times t during or after the execution of L: if A holds at all times from the initiation of L, up to and including the time of the last communication before time t, then

a) C holds at time t, and
b) if L terminates then q holds for the final state.

As with other proof systems of this type, this proof system is designed to specify and verify only safety properties. The observable and non-observable entities are exactly those of [21] and all of them are specified in order to make the proof system compositional. After processes have been composed by parallel composition, the internal communication channels can (by using an operator) be hidden from the resulting network. Thus, the proof system can be used to develop a modular design methodology with network abstraction. However, a formal specification language is yet to be developed, and in general it remains to be seen how convenient it is to use any proof system which requires the waiting times of processes to be specified.

4 Real-time Languages and their Semantics

Formal models of real-time programs are still at an early stage of development and a great deal of work remains to be done on aspects such as designing language constructs for real-time use, and developing proof systems and associated semantic models. Given the complexity of many real-time systems, this is a difficult objective and various aspects of real-time programming are therefore being investigated in isolation. At the present time, the development of specification and verification systems for real-time programs has been made either with simplifying assumptions about the semantics of existing language constructs, or by extending a language

which already has a well-established semantics with new constructs for expressing real-time. In this section we discuss the work in real-time programming which has placed emphasis on issues other than specification and verification.

Recent work on CSP-like languages [3,4,18,41] has made it possible to consider a transformational approach to concurrent programming. The underlying semantic models have been designed to have elegant algebraic properties which allow simple definitions of program equivalence, precedence in the distribution of operators, etc. Although this approach has the disadvantage that the initial specification must itself be a program, it is very convenient to use because the program calculus does not involve the semantic objects on which the algebra is based. A very interesting question would be to see if real-time programs can be developed by algebraic transformation. We describe three algebraic models of certain real-time versions of CSP.

In an algebraic model of CSP, a process is described in terms of its communications (or *events*) with its environment [3,18]. A *trace* of the process is a finite sequence of communications in which the process has participated upto a certain time. The semantics of a process is determined by the set of its traces and the future behaviour of the process after it has engaged in a particular trace. For real-time versions of CSP (obtained by introducing "**wait** d" or "**delay** d" commands), the semantics of a process must use an explicit denotation of time along with the trace and future behaviour. The future behaviour of a process in untimed CSP when it has engaged in a trace s specifies whether the process *diverges* (ie, engages in internal actions indefinitely) or not after engaging in s. This information is not sufficient to describe the real-time behaviour of a process [41]. If the process does not diverge after the trace s, it is necessary to know when the process will be *stable*, ie ready to participate in a communication. Thus the stability of a process should be timed in a real-time behaviour of the process. Time in this model is global, as seen by an external observer, and continuous. A timed CSP process is modelled by a set of pairs (s, α) where s is a *timed trace* (of *timed events*) and α, called the stability value for the trace, is the time at which the process is guaranteed to be stable after engaging in s. To distinguish between the times when a process actually does communicate and the times when it is ready to communicate, each event a has an associated unique event $\hat{a}$ which occurs exactly when the process becomes ready to participate in a.

To avoid process behaviours in which an unbounded number of events may occur in a finite time, a 'delay constant' δ is introduced in the semantic model: the prefix construct "$a \rightarrow P$" behaves like P only after time δ from its participation in a. Also, for similar reasons, a recursive process can participate in an observable event only after time δ from making a recursive call.

Unlike the model in Koymans et al [28], this model distinguishes between deadlock and divergence. The semantic domains of (s, α) pairs are structured as complete metric spaces. The semantic operators (including the problematic operator permitting 'hiding' of events) are continuous and all program combinators except recursion distribute over nondeterministic choice.

A simple version of Hoare's trace model of CSP [18] (i.e. without failure or

divergence) was extended by Zwarico and Lee [49] using timed events [41] to develop a timed trace model of deterministic CSP (without internally controllable choice). The algebraic properties of CSP in this model are not as rich as those of Reed and Roscoe[41]. However, the execution times of actions are more realistically modelled by associating intervals with the prefix construct, instead of a single fixed time. Time in this model is the domain of nonnegative integers. Every process has as its local time values from a subset of this time domain, including 0.

The algebraic model of CSP with "**wait** t" developed by Gerth and Boucher [10] is based on the failures model of CSP [3]. The time domain is any total order with a least element. In the untimed failures model of CSP, a *failure* of a process is a pair (s, X) where s is a trace and X is a *failure set* – the set of events in which the process will refuse to participate after engaging in s. The failures model adequately captures the notion of deadlock. The timed semantic model [10] uses relations between times and events, ie sets of (time, event) pairs so that if a process refuses to participate in an event the time of that refusal is known. This guarantees that whenever some actions are possible, at least one of these actions is taken within an *a-priori* known and bounded time interval. This precludes using an interleaving implementation of the system in which actions may be arbitrarily delayed. The semantics is fully abstract and all the process operators and the semantic operation of hiding are continuous.

Languages

CSP and related languages have been used to study the particular problems of real-time semantics because they are already equipped with several well understood models for non-real-time distributed programs. However, practical work in real-time programming has traditionally demanded less spartan languages, comparable to those used for other kinds of large-scale critical programs. The early languages used in real-time systems, such as Jovial, have become inappropriate now that there is better understanding of programming techniques and of real-time problems. A variety of other languages have been used for real-time programming (eg, RTL/2, CORAL 66 and Pearl) but there are probably even more systems where the real-time control commands have been provided by operating system features while the functional (and very often highly numerical) computation has been done in languages like FORTRAN.

A major response to the need for a real-time programming language was provided by Ada, which was designed specifically for embedded (real-time) systems. Being a large language, formal descriptions of even incomplete versions of Ada are large and there is difficulty in modelling its real-time features. A solution to this is to incorporate the real-time constructs of Ada into other smaller languages like CSP: this permits these constructs to be examined in relative isolation (see eg [20,28]). Even so, some of the more elaborate real-time features (such as explicit process termination, the use of priorities, etc) have not been incorporated into existing models.

It is of course true that some of the difficulty with modelling real-time features comes from the fact that we have been considering imperative languages with asynchronous events where all computations take some finite time. If the actions

of a real-time system can be assumed to take no time at all, ie with no delay between external events and responses, the system becomes *synchronous* and can be programmed in a deterministic language. A number of languages have made these assumptions: eg, ESTEREL [2],LUSTRE [5] and SIGNAL [12]. ESTEREL is an imperative language with a semantics defined as a set of rewrite rules. An ESTEREL program is compiled by converting it into a finite automaton which is implemented in some imperative language on a single processor system. LUSTRE and SIGNAL are similar synchronous data-flow languages which are best suited for problems that have few control states. A good account of the important differences between these languages has been given by Berry et al [2].

5 Scheduling Real-time Programs

Unlike specification or implementation level descriptions of programs, where each detail must be precisely defined, studies of scheduling are usually based on more global program properties. In many cases, a program is viewed only as consisting of a number of processes, each of which is a single schedulable entity, with no internal detail and with no communication with other processes. More recent studies have considered synchronization and intercommunicating processes, but again at a gross level. However, in contrast to this apparent lack of precision, the modelling of system resources is altogether more exact than is typical in semantic models. We shall first describe a simple program model and then consider how this can be related to the scheduling of resources in a real-time system.

A concurrent real-time program usually has a number of cyclic and sporadic processes: the cyclic processes need to be executed at regular intervals and the sporadic processes on demand but with some minimum time between successive demands. Without much loss of generality, *all* the processes can be considered to be cyclic if the sporadic processes are treated as requiring repeated execution with the specified minimum interval between executions. Let a program have n such processes, each process P_i making a request for C_i units of processor time every T_i time units. In a *hard real-time* system, a *valid* schedule must guarantee that, for some implementation, each process receives its processing requirement C_i within the interval T_i; an implementation in which this is achieved is called *feasible*.

The simplest way to implement such a program would be to use a system with n processors and to assign one for each process, ie to have a maximally parallel implementation. This however is rarely practicable, both because of cost and because the decomposition of a program into a number of processes is often done to simplify the design rather than to distribute the processing load uniformly. When a maximally parallel implementation is not possible, techniques are required for implementing the processes on a limited number of processors and for scheduling them to meet their real-time requirements.

The degree of flexibility available for scheduling is limited by the interdependence of processes. If the real-time program consists of processes that interact closely with each other, decisions about their scheduling will be constrained by the needs of synchronization, for example because processes sending messages must

be executed before those receiving the messages can be executed. If a process can undertake one of several actions and the choice between them is nondeterministic, scheduling the processes is considerably more difficult.

The early work on process scheduling assumed that processes were completely independent and that scheduling decisions could be taken without regard for program structure. For a single processor system, this reduces the problem to multiprogram scheduling. In some seminal work, Liu and Layland [34] showed that the worst-case scheduling problem arose when all these processes made their requests at the same time and they considered scheduling techniques for meeting this demand. One technique is to assign priorities to processes and to schedule them pre-emptively (ie, to suspend execution of a lower priority process when there is a request from a higher priority process). An effective way of assigning priorities to processes is to do so in rate-monotonic order, ie giving the process with the smallest value of T_i the highest priority, and so on. In fact, such a priority-based schedule is called *optimal* because if it does not provide a feasible solution, ie one in which each process receives its processing requirement, then no such solution exists.

In many cases, the processing requirement C_i for process i must be provided within a deadline D_i which is smaller than T_i; deadlines can also be used for priority assignment. Dertouzos [6] showed that dynamic priority assignment in order of D_i also optimal and Mok [39] showed that this was true as well if the order chosen was that of the least slack ($T_i - C_i$) or of any consistent combination between the earliest deadline and least slack order. The deadline can be extended to $2T_i$ in some cases and Moitra [37] has shown how feasible assignments of priorities to processes can be determined; other work [25] describes how response times for simple processes can be computed.

The repetition time T_i of process P_i is determined by the frequency with which some external process sends its events while the computation time C_i depends on the time taken to complete the processing requirement for the process, ie on the speed of the processor. When a feasible implementation is not possible for some processor, there are other possibilities: to use a faster processor, or to increase the number of processors. With a faster processor, the value of each C_i may be reduced to the point where it is possible to find a valid optimal schedule. When there is more than one processor, a number of new possibilities must be considered. At one (somewhat unrealistic) extreme, the number n of processes may be smaller than the number m of processors and more than one processor may be used for successive partially complete executions of one process. But it is altogether more likely that $m < n$ and there are questions of how the processes should be partitioned between the processors, and what the smallest value of m is for a valid schedule. Unfortunately, for all but trivial cases, the problem of deciding whether a valid schedule exists is NP-hard [32,33].

In most real-time programs, processes are *not* independent and schedules must take account of inter-process synchronization and communication, and the need for resource allocation. A simple requirement here is for mutual exclusion between processes and Mok has shown that unless all process execution times upto the point of synchronization are known, it is not possible obtain a valid schedule for a single or a multiple processor system [39]. And where semaphores are used to

ensure mutual exclusion, the problem of deciding whether there is a valid schedule is NP-hard [39].

Though general solutions to these problems are not accessible, solutions for restricted classes of problem can be obtained. Leinbaugh [32] considered a model in which processes could access input-output devices and compete for resources on a FIFO basis but must have a very simple sequential structure. Mok [38] has shown that if a feasible schedule is possible for a program using monitors [19] and rendezvous [45] then it can be achieved using an underlying, on-line mechanism (a so-called *kernalized* monitor). Basically, these results suggest that restrictions are needed both over process structure and over the use of communication mechanisms if valid schedules are to be found.

The scheduling techniques considered so far are based on static analyses of programs and there are cases where these lead to solutions that are more conservative than necessary; moreover, as seen above, many of the static scheduling problems do not have polynomial-time solutions. An alternative method of scheduling is to use heuristic algorithms for dynamic scheduling and to use measures like penalty functions or weighted success ratios to guide the choice. Zhao, Ramamritham and Stankovic [47,48] have studied how such algorithms can be used for resource scheduling in distributed hard-real-time systems.

6 Conclusions

Practical real-time systems have been in use for well over two decades, longer therefore than commercial timesharing and multiprocessing systems and far longer than distributed systems. However, they have been of little theoretical interest until recently. One reason for this is that they have often been considered to be just specially engineered versions of concurrent systems. Another reason may be that it is only in the last decade that studies of concurrency have reached the level of maturity needed for the major new problems of real-time systems to be tackled.

Rather than the division between imperative and functional programming which is traditional in other areas of programming, the semantics of real-time systems seem to be more appropriately divided between *synchronous* and *asynchronous* models, and between systems with unbounded resources and those with limited resources. If the resources of the system are sufficient for them to be no constraint on the speed of execution, a real-time system can be considered to be synchronous and programs can be written in a simple deterministic language [2] and implemented on a single processor. When execution times are not negligible but resources are unlimited, maximum parallelism models [20,41] provide a good representation of real-time systems. In both cases, there is no need for scheduling decisions about the allocation of resources.

It is not unreasonable to claim that there are many systems whose execution times *are* significant and whose resources, including the number and speed of the processors, are limited to the extent that their use must be scheduled. In fact, most current real-time systems fall into this category because their resources are limited either by cost or by environmental constraints on the weight, power dissipation, etc.

It is for such systems that the semantics must model limited resource executions and it is here that semantics and scheduling studies are most closely connected.

The pleasing aspect of the recent work in real-time systems is that it appears to be converging towards a common direction. Requirement definition, specification and semantic models are being extended to consider problems that are well-known in real-time programming. There is better understanding of the semantic implications of real-time programming features in languages and more concern with representing mechanisms such as priorities and timeouts. Scheduling models are also considering more realistic programs with features that are similar to those in programming languages. This has promise for the design of new programming language features and for development methods that will result in programs with functional correctness and guaranteed real-time properties.

References

[1] A. Bernstein and P.K. Harter, Jr. Proving real-time properties of programs with temporal logic. In *Proceedings 8th Annual ACM Symposium on Operating Systems Principles*, pages 1–11, 1981.

[2] G. Berry, P. Couronne, and G. Gonthier. *Synchronous Programming of Reactive Systems: An Introduction to ESTEREL.* Technical Report 647, INRIA, Sophia Antipolis, 1987.

[3] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[4] S.D. Brookes and A.W. Roscoe. An improved failures model for communicating processes. In *Lecture Notes in Computer Science 197*, pages 281–305, Springer-Verlag, Heidelberg, 1985.

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE, a declarative language for real-time programming. In *Proceedings 14th ACM Symposium on Principles of Programmming Languages*, 1987.

[6] M. Dertouzos. Control robotics: the procedural control of physical processes. In R.E.A. Mason, editor, *Information Processing 83*, North-Holland, Amsterdam, 1974.

[7] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[8] R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings Symposiain Applied Mathematics 19, American Mathematical Society, Providence*, pages 19–32, 1967.

[9] N. Francez, D. Lehmann, and A. Pnueli. A linear history semantics for languages for distributed programming. *Theoretical Computer Science*, 32:25–46, 1984.

[10] R. Gerth and A. Boucher. *A Timed Failures Model for Extended Communicating Processes*. Technical Report TR.4-4(1), Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, 1987.

[11] O. Grümberg, N. Francez, J.A. Makowsky, and W.-P. de Roever. A proof rule for fair termination of guarded commands. In J.W. de Bakker and J.C. van Vliet, editors, *Algorithmic Languages*, pages 399–416, North-Holland, Amsterdam, 1981.

[12] P. le Guernic and A. Benveniste. *Real-Time, Synchronous, Data-Flow Programming: The Language SIGNAL and its Mathematical Semantics*. Technical Report 620, INRIA, Rennes, 1986.

[13] V.H. Haase. Real-time behavior of programs. *IEEE Transactions on Software Engineering*, SE-7(5):494–501, 1981.

[14] I.J. Hayes (Editor). *Specification Case Studies*. Prentice-Hall, London, 1987.

[15] M. Hennessy and G.D. Plotkin. Full abstraction for a simple parallel programming language. In *Lecture Notes in Computer Science 74*, pages 108–121, Springer-Verlag, Heidelberg, 1979.

[16] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[17] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[18] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International U.K., 1985.

[19] C.A.R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[20] J. Hooman. *A Compositional Proof Theory for Real-Time Distributed Message Passing*. Technical Report, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, 1987.

[21] C. Huizing, R. Gerth, and W.-P. de Roever. Full abstraction of a real-time denotational semantics for an OCCAM-like language. In *Proceedings 14th ACM Symposium on Principles of Programmming Languages*, pages 223–238, 1987.

[22] INMOS Ltd. *The Occam Programming Manual*. Prentice-Hall International, U.K., 1983.

[23] F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12:890–904, 1986.

[24] C.B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, London, 1986.

[25] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[26] M. Joseph and P. Pandya. *Specification and Verification of Total Correctness of Distributed Programs.* Technical Report 96, Department of Computer Science, University of Warwick, Coventry, 1987.

[27] R. Koymans. *Specifying Message Passing and Real-Time Systems with Real-Time Temporal Logic.* Technical Report, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, 1987.

[28] R. Koymans, R.K. Shyamasundar, W.-P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. In *Lecture Notes in Computer Science 193*, pages 167–190, Springer-Verlag, Heidelberg, 1985.

[29] R. Koymans, J. Vytopil, and W.-P. de Roever. Real-time programming and asynchronous message passing. In *Proceedings 2nd ACM Symposium on Principles of Distributed Computing*, pages 187–197, 1983.

[30] L. Lamport. "Sometime" is sometimes "not never": on the temporal logic of programs. In *Proceedings 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, 1980.

[31] L Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[32] D.W. Leinbaugh. Guaranteed response times in a hard-real-time environment. *IEEE Transactions on Software Engineering*, 6(1):85–91, 1980.

[33] J.Y.-T. Leung and M.L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[34] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprocessing in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[35] R. Milner. *A Calculus of Communicating Systems, Lecture Notes in Computer Science 92.* Springer-Verlag, Heidelberg, 1980.

[36] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[37] A. Moitra. Scheduling of hard real-time systems. In *Lecture Notes in Computer Science 241*, pages 362–381, Springer-Verlag, Heidelberg, 1986.

[38] A.K. Mok. The design of real-time programming systems based on process models. In *Proceedings IEEE 1984 Real-Time Systems Symposium*, pages 5–17, Austin, Texas, 1984.

[39] A.K. Mok. *Fundamental Design Problems Of Distributed Systems for the Hard Real-Time Environment.* Technical Report MIT/LCS/TR-297, Massachusetts Institute of Technology, 1983.

[40] A.K. Mok. SARTOR - a design environment for real-time systems. In *COMPSAC 85*, pages 174–181, 1985.

[41] G.M. Reed and A.W. Roscoe. A timed model for Communicating Sequential Processes. In *Lecture Notes in Computer Science 226*, pages 314–323, Springer-Verlag, Heidelberg, 1986.

[42] A. Salwicki and T. Müldner. On the algorithmic properties of concurrent programs. In *Lecture Notes in Computer Science 125*, pages 169–197, Springer-Verlag, Heidelberg, 1981.

[43] A.U. Shankar and S.S. Lam. Time-dependent distributed systems: proving safety, liveness and real-time properties. *Distributed Computing*, 2:61–79, 1987.

[44] M. Shaw. *A Formal System for Specifying and Verifying Program Performance*. Technical Report CMU-CS-79-129, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1979.

[45] United States Department of Defense. *Reference Manual for the Ada Programming Language*. Technical Report ANSI/MIL-STD-1815A, 1983.

[46] B. Wegbreit. Verifying program performance. *Journal of the ACM*, 23(4):691–699, 1976.

[47] W. Zhao, K. Ramamritham, and J.A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, 36(6):949–960, 1987.

[48] W. Zhao, K. Ramamritham, and J.A. Stankovic. *Scheduling Tasks with Resource Requirements in Hard Real-Time Systems*. Technical Report, Department of Computer and Information Science, University of Massachusetts, Amherst, 1985.

[49] A. Zwarico and I. Lee. *A Syntax and Semantics for Deterministic Real-Time Computing*. Technical Report, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, 1987.

[50] J. Zwiers. *Compositionality, Concurrency and Partial Correctness: Proof theories for networks of processes, and their connection.* PhD thesis, Technical University of Eindhoven, Eindhoven, 1988.

[51] J. Zwiers, W.-P. de Roever, and P.E.M de Boas. Compositionality and concurrent networks: soundness and completeness of a proof system. In *Lecture Notes in Computer Science 194*, Springer-Verlag, Heidelberg, 1985.